Solving the Santa Claus Problem

A Comparison of Various Concurrent Programming Techniques

Jason L. Hurt & Matt Pedersen, University of Nevada, Las Vegas
What did we do?

• Implemented ‘The Santa Claus Problem’ in a number of different programming paradigms.
Why did we do that?

- Investigate a number of concurrent programming techniques in order to better understand issues related to:
  - Readability
  - Writability
  - Reliability
  - Error Handling
How did we do it?

• Write code…
  ✦ Consider
    ▪ Readability/Writability
      • What adds/detracts to/from the readability/writability
      • How does error handling affect this
    ▪ Reliability
      • How do we know it works
      • Model checking possibilities
  ✦ Count lines and compare
The Santa Claus Problem
[Originally by John Trono]

• Santa Claus sleeps at the North Pole until awakened by either all of the nine reindeer, or by a group of three out of ten elves.

• If awakened by the group of reindeer, Santa harnesses them to a sleigh, delivers toys, and finally unharnesses the reindeer who then go on vacation.

• If awakened by a group of elves, Santa shows them into his office, consults with them on toy R&D, and finally shows them out so they can return to work constructing toys.
Additional Constraints

• A waiting group of reindeer must be served by Santa before a waiting group of elves.

• Since Santa’s time is extremely valuable, marshaling the reindeer or elves into a group must not be done by Santa.
Correctness of a Solution

- **Message ordering**: ensuring that events happen in correct order at the right time.
- **Priority**: ensuring that the group of Reindeer have priority over any Elf groups that may be waiting at the time.
- **Self-Organization**: Santa cannot marshal a group of Elves or Reindeer; these groups must organize among themselves without help from a Santa thread or process.
- **Synchronization**: synchronization between various processes.
- The usual freedom from deadlock, livelock, and starvation.
Example: Elf Message Ordering

1. Elf <id>: need to consult santa, :(  
2. Santa: Ho-ho-ho ... some elves are here!  
3. Santa: hello elf <id> ...  
4. Elf <id>: about these toys ... ???  
5. Santa: consulting with elves ....  
6. Santa: OK, all done - thanks!  
7. Elf <id>: OK ... we’ll build it, bye ... :(  
8. Santa: goodbye elf <id> ...  
9. Elf <id>: working, :)  

[Note, Reindeer Messages can be interspersed between elf messages]
Example: Reindeer Message Ordering

1. Reindeer <id>: on holiday ... wish you were here, :) 
2. Reindeer <id>: back from holiday ... ready for work, : 
3. Santa: Ho-ho-ho ... the reindeer are back! 
4. Santa: harnessing reindeer <id> ... 
5. Santa: mush mush ... 
6. Reindeer <id>: delivering toys ... la-di-da-di-da-di-da-di-da, 
7. Santa: woah ... we're back home! 
8. Reindeer: <id>: all toys delivered ... want a holiday, :( 
9. Santa: un-harnessing reindeer <id> ...
The following processes are required:
- 10 elves
- 9 reindeer
- 1 Santa

These processes might be needed for synchronization and self-organization reasons:
- Processes to implement barriers
- Processes to implement waiting rooms etc.
The Paradigms & Models

- **Shared Memory (Threads)**
  - Pthreads in C
  - Java and Groovy
  - .NET Threading library
  - Polyphonic C#

- **Message Passing**
  - MPI

- **Process oriented**
  - JCSP
  - Occam (Thanks to Peter Welch/Matt Pedersen)
  - Groovy (Thanks to Jon Kerridge)
And now for something …

- The next (many) slides will consider a number of issues dealing with
  - synchronization, priority, etc in the different programming models
  - What is the issue
  - How does it affect the code
• Issue: Synchronization
  ✦ For thread synchronization, we define our own barrier type using a mutex and a condition variable from the pthread library.
  ✦ Santa code that uses the barriers:

    */ notify elves of “OK” message */
    AwaitBarrier(&elfBarrierTwo);
    /* wait for elves to say “ok we’ll build it” */
    AwaitBarrier(&elfBarrierThree);
C & pthraeds

**Issue: Priority**

- Mutexes and Condition Variables used for Reindeer over Elves priority: 😘

```c
pthread_mutex_lock(&santaMutex);
pthread_cond_signal(&santaCondition);
pthread_mutex_unlock(&santaMutex);
```

- A shared memory counter must be used to keep track of missed notifications.
Java Threads

• Issue: Synchronization/Self organization
  - Partial (and full) barrier
    - There are no barriers in the standard Java language
  - Solution: CyclicBarrier 😊
    - CyclicBarrier [library in Java 1.5] for thread synchronization eliminates the need for explicit shared state among synchronizing threads
    - Re-entrant - call to reset will allow the barrier to be used again
Java Threads

• Issue: Priority
  ♦ Priority is achieved via `wait/notify`.

• The `notify` method is asynchronous, it will complete even if a Thread with a corresponding `wait` call is not currently ready to receive the notification:

```java
synchronized (m_santaLock) {
    m_santaLock.notify();
    notifiedCount++;
}
```

[corresponding code exists on the Santa side]
Java Threads

- Issue: Spurious Wakeups.
  - Due to spurious wakeups, JVM is permitted to remove thread from wait sets without explicit instructions, which causes extra logic around calls to wait:

    ```java
    while (!<some condition>) {
    try {
        obj.wait();
    }
    catch(InterruptedException ie) { }
    }
    ```

  - Where `<some condition>` is set by notifying thread.
Issue: Synchronization

- Very similar to mutex and condition variable programming with pthreads. We build our own Barrier type that can be used for synchronization around Monitors.

- Same problem as Java threads and pthreads, the notification method, `Monitor.pulse`, is asynchronous, so threads must share state for the Santa thread to check for lost notifications.
Polyphonic C# (Chords)

- **Issue: Synchronization**
  - Associates a code body with a set of method headers. The body of a chord can only run once all of the methods in the set have been called.
  ```csharp
  int f(int n) & async g(int m) {
    ...
  }
  ```
  - A wait/notify mechanism that can prioritize notifications can be implemented with shared memory if chords are available to the programmer.
C & MPI

• Issue: Synchronization
  - Groups, or subsets of processes, can be formed at runtime, so we create a group that consists of Santa and all of the Reindeer: 😊

```c
MPI_Group_incl(groupWorld, TOTAL_REINDEER+1, santaReindeer, &groupSantaReindeer)
```

```c
//create communicator based on subgroup
MPI_Comm_create(MPI_COMM_WORLD, groupSantaReindeer, &commSantaReindeer);
```
C & MPI

- Issue: Synchronization (continued)
  - MPI_Barrier:

    \[
    \begin{align*}
    &\text{// wait for all reindeer to say “delivering toys”} \\
    &\text{mpiReturnValue = MPI_Barrier(commSantaReindeer);} \\
    &\text{CHECK_MPI_ERROR(globalRank, mpiReturnValue);} \\
    &\text{printf("Santa: woah . . . we’re back home!\n");}
    \end{align*}
    \]

  - Indirect synchronization using MPI_Send/MPI_Recv:

    \[
    \begin{align*}
    &\text{mpiReturnValue = MPI_Recv(&recv, 1, MPI\_INT,} \\
    &\text{MPI\_ANY\_SOURCE,} \\
    &\text{elfTag, MPI\_COMM\_WORLD,} \\
    &\text{&status);}
    \end{align*}
    \]
• **Issue: Priority**
  - Santa probes to see if the reindeer are ready before servicing a group of elves or reindeer with an asynchronous MPI_Iprobe:

    ```c
    int checkReindeerFlag = 0;
    mpiReturnValue = MPI_Iprobe(REINDEER_QUEUE_PROC, santaNotifyTag, MPI_COMM_WORLD, &checkReindeerFlag, &status);
    ```

  - We use separate processes to gather the deer or the 3 of 10 elves
    - REINDEER_QUEUE_PROC, ELF_QUEUE_PROC
Barriers with Channels (JCSP). Implemented barriers for synchronizing Santa and a group of 3 Elves or Santa and the Reindeer using 2 shared channels.

- MyBarrier holds the reading end of the channels and Sync holds the writing end of the channels, only when all members of the barrier have sent their first message will a process start to send its second message to the reading end of the barrier:

```java
// wait for Elves to say “about these toys”
new Sync(outSantaElvesA, outSantaElvesB).run();
outReport.write("Santa: consulting with Elves . . .\n");
```
Issue: Synchronization (Continued)

- Santa and the Reindeer use an array of One2OneChannel<Int> types for synchronization.
  - Santa code:
    ```java
    //unharness a Reindeer
    channelsSantaReindeer[id - 1].out().write(0);
    ```
  - Reindeer code:
    ```java
    //wait to be unharnessed
    inFromSanta.read();
    ```
• Issue: Priority
  
  For priority, the JCSP version uses an alternation which waits for guarded events which can be prioritized: 😍

```
final Guard[] altChans = { inFromReindeer, inKnock };
final Alternative alt = new Alternative(altChans);
switch (alt.priSelect()) {
  //...santa logic here
}
```
So Far … So Good

- We have seen examples of how to deal with
  - Synchronization
    - Full Barrier
    - Partial Barrier
  - Priority
  - Language Specific Curiosities
    - Lost notifications
    - Spurious wakeups
Readability/Writability Factors

- Readability and Writability are impacted by
  - Code to deal with undesirable concurrency behavior
    - Spurious wakeups, lost notifications
  - Code Coupling
    - Shared state
    - Message tagging
  - Error handling
    - More to come about that …. 
  - Code to implement prioritized notifications
    - PriALT
    - MPI_Iprobe
Error Handling (Java)

- Checked exceptions in Java often require code that is quite verbose, even for simple logging of the exception. So a call to `CyclicBarrier.await()` looks like this:

```java
// notify elves of "OK" message
try {
    m_elfBarrierTwo.await();
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (BrokenBarrierException e) {
    e.printStackTrace();
}
```
Error Handling (Groovy)

- Use **closures** for exception handling logic and thread related operations and a separate method takes the thread library call logic and wraps it in the exception handling logic:

  ```java
  //notify santa of "ok" message
  performOperation(barrierAwait(m_elfBarrierThree))
  ```
Both the .NET threading library and the pthread library support errors, the
languages do not force handling of the errors so the code is less verbose.

In C# all exceptions are unchecked and the pthread library call return error codes
which we (can) silently ignore.

Error Handling (C#/pthread)
Error Handling (MPI)

• The MPI library does not force error handling, but due to the distributed nature of MPI it is good practice to check for errors to MPI library calls. We define a macro `CHECK_MPI_ERROR` that will handle the errors:

```c
#define CHECK_MPI_ERROR(rank, errorId) { \
  if(errorId != MPI_SUCCESS) { \
    printf("Global Rank #\%d exiting, mpi error code: %d\n", rank, errorId); \
    MPI_Finalize(); \
    return -1; \n  } \n} \
```

[Note, Errors always imply termination, which can put the machine in an undesirable state]
Error Handling (JCSP)

- The parts of the JCSP library that we used did not declare any checked exceptions, so there is no error handling code here.
- Occam/JCSP error handling on concurrency errors: poison
Error Handling (General)

- Seems that most error handling is language specific (try/catch etc)
- Concurrency errors often just terminates the program
  - Poison in process oriented language
  - Ctrl+c & “clean the virtual parallel machine” with MPI
  - Crash the program in Java/C etc.
• Shared state increases coupling and makes re-factoring more challenging 😞
• JVM spurious wakeups are nasty 😞
• Java Thread.notify, pthread condition variable, and .NET Monitor.pulse may cause lost notifications, which forces shared state to be used among threads 😞
• MPI synchronous receives are nicer for synchronizing than notify since the sent message does not get lost. 😊
Readability/Writability Results

- JCSP channels increase modularization and message integrity over MPI, must have explicit reading or writing end of a channel.

- Error handling is non-trivial in all cases.
Reliability

- Hard to reason about concurrent code. 😞
- We could model check the code 😞
  - CSP & FDR
  - SPIN
  - ...

Model Checking

• JCSP/occam maps to CSP which can be model checked.
  - Might turn into machine assisted verification.

• MPI-Spin can be used to check various aspects of MPI.
  - Has less of a correspondence to MPI than CSF has to occam and JCSP
## Line Count Comparison

<table>
<thead>
<tr>
<th>Category</th>
<th>SM</th>
<th>DM</th>
<th>PO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C#</td>
<td>C</td>
<td>Java</td>
</tr>
<tr>
<td>Total</td>
<td>642</td>
<td>420</td>
<td>564</td>
</tr>
<tr>
<td>Synchronization/Communication</td>
<td>48</td>
<td>49</td>
<td>46</td>
</tr>
<tr>
<td>Prevent Race Condition</td>
<td>14</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Exception/Error Handling</td>
<td>35</td>
<td>0</td>
<td>177</td>
</tr>
<tr>
<td>Custom Barrier Implementation</td>
<td>42</td>
<td>35</td>
<td>N/A</td>
</tr>
<tr>
<td>GUI</td>
<td>145</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

SM = Shared Memory, DM = Distributed Memory, PO = Process Oriented
So what did we learn?

• Code that requires heavy synchronization can be done better in MPI and even better in occam or JCSP than with threads.
• Prioritization is made easier with the prioritized alternation construct.
• Error handling is non-trivial in all cases.
More Santa

- Jon Kerridge’s Groovy Fringe presentation
- Peter Welch’s occam-\(\pi\) mobile processes Fringe presentation
- [www.santaclausproblem.net](http://www.santaclausproblem.net) for all the code
Future Work

- Different problems
- More models/languages, Shared Transactional Memory
- Feasibility/ease of model checking for the various models
Questions